SANDIA REPORT

SAND98–0074 • UC–905
Unlimited Release
Printed January 1998

Developing Robotic Behavior Using a Genetic Programming Model

R. J. Pryor

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A04
Microfiche copy: A01

SAND-98-0074
Unlimited Release
Printed January 1998

Distribution
Category UC–905

Developing Robotic Behavior Using a Genetic Programming Model

R. J. Pryor
Program Management Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1109

Abstract

This report describes the methodology for using a genetic programming model to develop tracking behaviors for autonomous, microscale robotic vehicles. The use of such vehicles for surveillance and detection operations has become increasingly important in defense and humanitarian applications. Through an evolutionary process similar to that found in nature, the genetic programming model generates a computer program that when downloaded onto a robotic vehicle's on-board computer will guide the robot to successfully accomplish its task. Simulations of multiple robots engaged in problem-solving tasks have demonstrated cooperative behaviors. This report also discusses the behavior model produced by genetic programming and presents some results achieved during the study

Intentionally Left Blank

Contents

Figures

Tables

Developing Robotic Behavior Using a Genetic Programming Model

Introduction

Developing tracking behaviors for autonomous, microscale robotic vehicles is becoming an increasingly important technical challenge. These vehicles are being considered for a variety of defense and humanitarian applications. For defense, they could be used to conduct surveillance and to gather information. As part of humanitarian efforts, the vehicles could be used to locate and remove land mines. These types of tasks require that the robotic vehicles have sophisticated tracking behaviors so that they can detect and maneuver around obstacles in their search for a source or target.

Operationally, it is envisioned that tens to hundreds of these microscale robots would be deployed to complete a given task. Each robot would have on-board electronics, including a small computer, ground-positioning and communication equipment, an obstacle detector, and some source-analysis capability. To enable movement, each robot would also have a motor, wheels, and a motor control system. Although the deployed robots would behave autonomously, each robot would communicate information with other robots during the task.

The types of tasks addressed in this report are limited to those necessary for locating a source. In a typical task, robots are initially distributed randomly in a field and given the goal of locating a source that is emitting some kind of signal (smell or sound). A robot's behavior takes the form of a computer program that provides instructions to the motor control system to direct the robot to move to the source location while navigating around obstacles. The language used in the computer program is not important because the program is compiled on another computer and then downloaded to the on-board computer of each robot deployed in the task. Several methodologies have been suggested to develop this behavior, ranging from thermodynamic analogy models to traditional guidance models. The model used and discussed in this report is called genetic programming. This model is different because the behavior program that was created was not written by a programmer. Instead, the program was created by another program, as is explained in this report.

In the sections that follow, we will describe this methodology in a logical step-wise fashion. The goal is to provide an overview of the whole process so the reader has a good understanding of how the problem of finding a source was solved.

Genetic Programming Model

Genetic programming is one of many types of genetic algorithms that use evolutionary or adaptive processes to solve practical problems. Holland's pioneering book, *Adaptation in Natural and Artificial Systems*, provides a general framework for such analysis. Many books have since been written on genetic algorithms, with Goldberg's *Genetic Algorithms in Search, Optimization, and Machine Learning* ranking among the best. Koza's *Genetic Programming* is the most informative source on the theory of genetic programming. This book is very well written, provides an excellent bibliography, and fills in much of the detail not provided in this report.

So what is genetic programming? Genetic programming is a methodology. When a computer program employs this methodology, it produces as output the source code of another computer program. This source code can then be compiled and executed. Unlike most computer programs, these programs are not written by a human programmer. These programs are said to evolve in a biological setting, with rules of natural selection and survival of the fittest playing an important part in their evolution.

Evolution occurs in discrete steps called generations. A generation is composed of a population of individuals, each of which is a complete computer program. The size of the population varies based on the problem; but hundreds, if not thousands, of programs are typical. Some programs (individuals) will be very effective at doing the prescribed task —some will not. Each program is scored for applicability, and its fitness is given a numerical score. The higher the fitness, the better the individual. The goal is to evolve the very best program that solves the problem of interest. In this application of genetic programming, we are trying to find the program that best guides our robots.

The solution strategy is to improve upon these programs by creating successive generations of more fit individuals. To create the next generation of individuals, genetic operators of selection, reproduction, crossover, and mutation are used. The purpose of selection is to choose an individual from the current population. In general, this individual will be better than most, but may not be the very best. Reproduction moves a selected individual directly into the next generation. Crossover uses the selection operator twice to select two parents from the current population that will be mated in some way to form an offspring that will be placed in the next generation. Mutation uses the selection operator once to choose an individual that will be mutated (changed) in some way and then placed in the next generation. The four genetic operators are discussed in more detail in later sections of this report.

The evolutionary calculation described above proceeds across many generations until a single individual is found that meets a convergence criteria. This program is then saved for use by the robots.

Program Representation

This section describes how the individual programs are represented within the genetic program, which is a program that employs genetic programming. The representation should allow complete flexibility in defining programs, yet it must also ensure that the performance of the genetic operations is not too cumbersome. A tree-like structure best meets these requirements.

The basic building block of a tree is called a *node*, with all nodes in the tree having the same fixed structure. The first element of a node specifies the node type, which can either be a function or a terminal. A *function node* performs a mathematical or boolean operation and generally has branches (nonzero pointers) that point to other nodes. The number of branches depends on the *kind* of function, e.g., add, subtract, multiply. A *terminal node* normally returns a value, does not have any branches (all pointers are zero), and terminates that section of the tree. Other elements within a node are a value position and pointers to other nodes. Consider the sample tree shown in Figure 1 below.

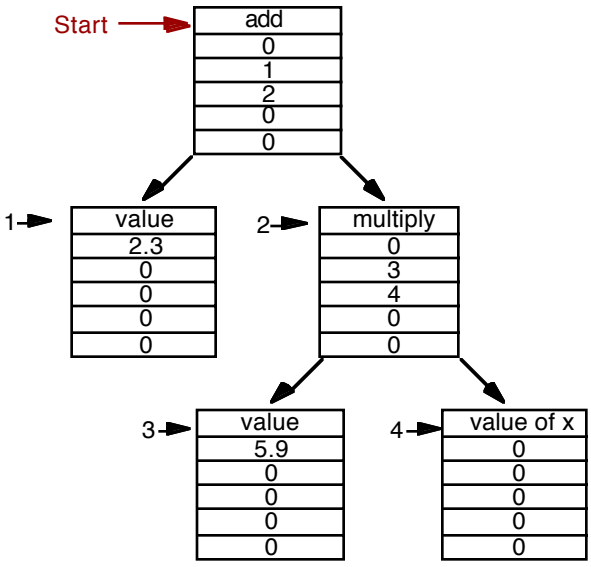

Figure 1. An example of a numerical tree.

This tree has five nodes and is three levels deep. The tree is evaluated by starting at its root, or top, and working downward until a terminal node is reached. A terminal node returns a value which is then processed upward in the tree.

To evaluate the sample tree, we begin at the first node denoted by "Start," which is a function node, whose kind is specified as *add*. This kind of function node points to two other nodes that will return values that will be summed by the add node. At pointer 1, there is a terminal node that returns a constant value of '2.3.' At pointer 2, there is a function node whose kind is *multiply*. This node points to two other nodes: one is a value node that returns the value '5.9,' and the other is a value node that returns the value of a global variable X. These two values will be multiplied by the multiply node, which will return the resultant to the add node above it. The tree is equivalent to the expression:

$$y = 2.3 + 5.9\ X$$

where *y* is the value returned by the root node at the top of the tree. The tree used in the robotics program has many more function and terminal types than in the sample tree, and the trees in the robotics program are also much larger.

Problem Definition

To evolve optimum tracking behaviors, we have developed a set of 90 problems that the robots must solve. At the start of each problem, the robots are cast in arbitrary positions onto a two-dimensional grid. Similarly, two walls and a target are cast onto this same grid also in arbitrary positions; the walls follow the grid lines in either the x or y directions. The goal for the robots is to learn to navigate around the walls and find the target. Importantly, the robots have no foreknowledge of either their own positions or the positions of the walls and the target.

Figure 2 illustrates a sample configuration at startup. The robots are represented by bugs[1], the target by a pile of cash, and the walls by heavy lines. The position of each robot is given by a coordinate pair (x,y) which are positive integers. A robot's orientation can be in one of four directions: north towards the top of the page, east towards the right, south towards the bottom, and west towards the left. The direction impacts the robot's sensing ability: a robot can only sense an obstruction if it is positioned in the direction the robot is facing. The single target has the coordinates (X_t,Y_t).

[1] Sandia calls these bug looking robots "Robugs."

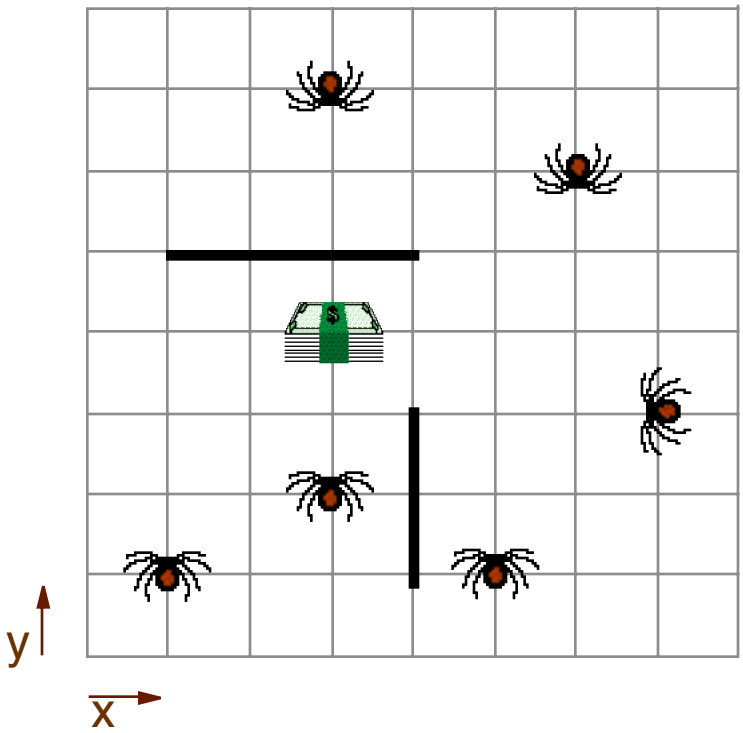

Figure 2. Geometry for robots, target, and walls.

There are some constraints that bound the problems, however. We assume that memory on the robot's on-board computer is limited, and therefore we store only four values of data. We also assume that communication between robots is limited to the two nearest neighbors. The data that can be communicated consist of positions and signal strengths. And because of assumed limits in the motor control system, only one movement instruction can be returned with each execution of the behavior program. This instruction allows the robot to move ahead one gridpoint or to turn to a new direction. Lastly, we assume that the signal emitted from the target is given by a one-over distance relationship. Given these conditions and constraints, the objective is to get each robot to the target.

Definition of Functions and Terminals

This section defines the set of functions and terminals that are used by our behavior programs. It is important to note that our selection of particular functions and terminals is not unique, nor is there any theory that indicates whether any one set is better than another. The only way to determine the effectiveness of a set is to try it out and see if it works.

The function and terminal set used in our robotic programming application consisted of 34 nodes, as listed in Table 1. There were 11 functions (Index 1–11) and 23 terminals (Index 12–34). It should be noted that all nodes return a value, even if they direct movement of the robot.

Table 1. List of Functions and Terminals

Index	Name	Description	Number of branches
1	ROOT	Root node. Returns the value from below unchanged.	1
2	ADD	Returns the sum of its two branches.	2
3	SUBTRACT	Returns the difference of its two branches.	2
4	MULTIPLY	Returns the product of its two branches.	
5	DIVIDE	Returns the division of its two branches. A test is made to ensure that a division by zero is not possible.	2
6	IFGTEQ	This is a conditional node. Branch one is evaluated and if greater than zero, this node returns the results of branch 2; otherwise, it returns the result of branch 3.	3
7	ABS	Returns the absolute value of its single branch.	1
8	COMPANG	Returns an integer value based on the point location of the x and y register values on the xy-plane. A value of 1 is returned if the point is nearest to the positive y-axis. A value of 2 is returned if the point is nearest to the positive x-axis. A value of 3 is returned if the point is nearest to the negative x-axis. A value of 4 is returned if the point is nearest to the negative y-axis. The branch value is ignored.	1
9	STOREX	Returns the value of its single branch but also stores this value in the x-register of the robot's local memory.	1
10	STOREY	Returns the value of its single branch but also stores this value in the y-register of the robot's local memory.	1
11	INT	Returns the nearest integer of the value of its	1

Index	Name	Description	Number of branches
		branch (e.g. 3.2 is converted to 3.0, and 3.6 is converted to 4.0).	
12	NEIGH1X	Returns the x-position of its first nearest neighbor.	0
13	NEIGH2X	Returns the x-position of its second nearest neighbor.	0
14	NEIGH1Y	Returns the y-position of its first nearest neighbor.	0
15	NEIGH2Y	Returns the y-position of its second nearest neighbor.	0
16	POSX	Returns its own x-position.	0
17	POSY	Returns its own y-position.	0
18	DIRECT	Returns its own direction (orientation). A value of 1 is returned if the robot is facing the positive y-direction. A value of 2 is returned if the robot is facing the positive x-direction. A value of 3 is returned if the robot is facing the negative y-direction. A value of 4 is returned if the robot is facing the negative x-direction.	0
19	SIG1	Returns the signal strength received by its first nearest neighbor.	0
20	SIG2	Returns the signal strength received by its second nearest neighbor.	0
21	SIG	Returns the signal strength that it is receiving.	0
22	VERTWX	When a robot is moving in the x-direction and strikes a barrier, the location of an assumed vertical barrier is stored in the local memory of the robot. This node returns the x-position of that barrier.	0
23	VERTWY	When a robot is moving in the x-direction and strikes a barrier, the location of an assumed vertical barrier is stored in the local memory of the robot. This node returns the y-position of that barrier.	0
24	HORZWX	When a robot is moving in the y-direction and strikes a barrier, the location of an assumed horizontal barrier is stored in the	0

Index	Name	Description	Number of branches
		local memory of the robot. This node returns the x-position of that barrier.	
25	HORZWY	When a robot is moving in the y-direction and strikes a barrier, the location of an assumed horizontal barrier is stored in the local memory of the robot. This node returns the y-position of that barrier.	0
26	RECALLX	Returns the value stored in the x-register of the robot's local memory.	0
27	RECALLY	Returns the value stored in the y-register of the robot's local memory.	0
28	TURNN	Directs the robot to turn north (positive y-direction). Returns a value of 1.0.	0
29	TURNE	Directs the robot to turn east (positive x-direction). Returns a value of 1.0.	0
30	TURNS	Directs the robot to turn south (negative y-direction). Returns a value of 1.0.	0
31	TURNW	Directs the robot to turn west (negative x-direction). Returns a value of 1.0.	0
32	TURNRGT	Directs the robot to turn right by 90 degrees. Returns a value of 1.0.	0
33	MOVE	Directs the robot to move one space point in the direction it is facing. If successful, the node returns a value of 1.0. If not successful, a barrier is assumed and the node returns a value of -1.0 and the barrier position is stored in local memory (see above).	0
34	VALUE	Returns the value stored in the behavior tree.	0

Tree Generation

From previous discussion you may recall that the genetic operators are used to create the next generation of behavior programs (trees) from the current generation, but you may be wondering how the first generation was created. The answer is that the first generation of trees is created randomly. To generate a tree, a recursive function is called that needs only to know the current level or position in the tree. We define level 1 as the top of the

tree or root. Level 2 is the level just below the root. Level 3 and the remaining levels follow.

The level number is initially set to zero. When the recursive function is called, it first increments this level number by one and then checks on its value. If the level number is equal to one, it inserts a ROOT node, defines a single new node, sets a pointer to it, and then calls itself pointing to the new node. On all other levels, the recursive function selects randomly from the functions and terminals, defines the appropriate number of new nodes and pointers, and then calls itself for every new node.

Two parameters control the size, or number of levels, of the tree. The parameter MINTREESIZE specifies the minimum number of levels a tree may have, while the parameter MAXTREESIZE specifies the maximum number of levels of a tree. These parameters are used in the following way. If the level number is less than MINTREESIZE, the recursive function will only select from the *function* node kinds (e.g., add, subtract). This ensures that at least one more level will be added. If the level number is equal to MAXTREESIZE, the recursive function will only select from the *terminal* node kinds. This ensures that no more levels are added to this part of the tree because terminal nodes do not have any branches. For all other level values, a random selection is made.

Fitness Evaluation

The most important aspect of the genetic programming methodology is the evaluation of tree fitness. Fitness provides the selection pressure that drives the programs to the desired behavior. The fitness of each tree in a population is evaluated separately. An evaluation involves testing a tree against a set of problems. After all problems have been run, a fitness score for the tree is determined.

A problem description consists of an initial configuration of robots, barriers, and target position. We select the configurations randomly using a random generator. Once these initial conditions are specified, a simulation calculation is performed. Robots move step-wise according to instructions provided by the behavior program. Each robot gets a chance to move before the next step is started. There are a finite number of steps in a simulation, and during each step a robot is permitted only a single movement (turn or move ahead). After the prescribed number of steps have elapsed, the positions of the robots are observed and the squared distance from each robot to the target is tallied. Once all problems are completed, a mean squared distance is computed and the fitness is determined from the expression below:

$$\textbf{fitness} = 1000.0 / \left(1.0 + \sqrt{\textbf{msd}} \right)$$

where **msd** is the mean squared distance. To prevent trees from becoming exceedingly large, we use a penalty function to reduce the fitness by a slight amount depending on the number of nodes in the tree. This penalty function is of the form:

$$\textbf{penalty} = \alpha\ (\textbf{number of nodes})$$

where α is a small constant.

Creating the Next Generation

The operators of reproduction, crossover, and mutation are used to create the next generation of individuals. Each operator works independently of the other operators, and the usage of each operator is determined by its assigned probabilities. The reproduction, crossover, and mutation operators use the selection operator to determine the individuals in the current population that will be acted upon.

The population size, POPSIZE, of each generation remains the same. To create the next generation, a loop over the necessary individuals is started. Within the loop, a random number is drawn and three probabilities are compared. The probability PROBREP determines the percentage of times within the loop that reproduction is used. Likewise, PROBCR and PROBMUT determine the percentage of times that crossover and mutation are used, respectively. The sum of the three probabilities is 1.0. From the random number and the three probabilities, an operator is selected that then creates one new individual in the next generation. When the required number of individuals is created, the loop is terminated.

Selection Operator

The selection operator is used to identify individuals in the current population for reproduction, crossover, or mutation operations. A tournament algorithm is used[2], which is easy to implement and is relatively fast. The algorithm works in the following way. NTOUR individuals are randomly selected from the population. The fitness values of these individuals are compared, and the individual with the highest fitness is the winner of the tournament; that is, the one selected. If two individuals are needed, as in the case of crossover, then the tournament is repeated. Note that the parameter NTOUR determines the distribution of individuals selected, and thus increasing its value moves distribution toward more elite individuals in the population. For example, if the value of NTOUR is equal to the population size, then only the most fit individual will be selected. Reducing the value of NTOUR allows more individuals to take part in forming the next generation

[2] Several different algorithms were considered, such as roulette wheel selection, but this worked well and it was fast.

Reproduction Operator

The reproduction operator is the simplest of the operators used. Reproduction makes an exact copy of a selected individual in the current generation and places it into the next generation. The fitness of this individual is saved so that the fitness value will not need to be recomputed.

Crossover Operator

The crossover operator is slightly more complicated than the reproduction operator. Crossover is done in three steps. First, the selection operator is called twice to select two individuals from the current population. Next, for each selected individual, a cutpoint is randomly selected among the nodes of its tree. Finally, the new tree is created by removing the cutpoint node and all nodes below it from the first tree and replacing them with the cutpoint node and all nodes below it from the second tree. Figure 3 illustrates this last step.

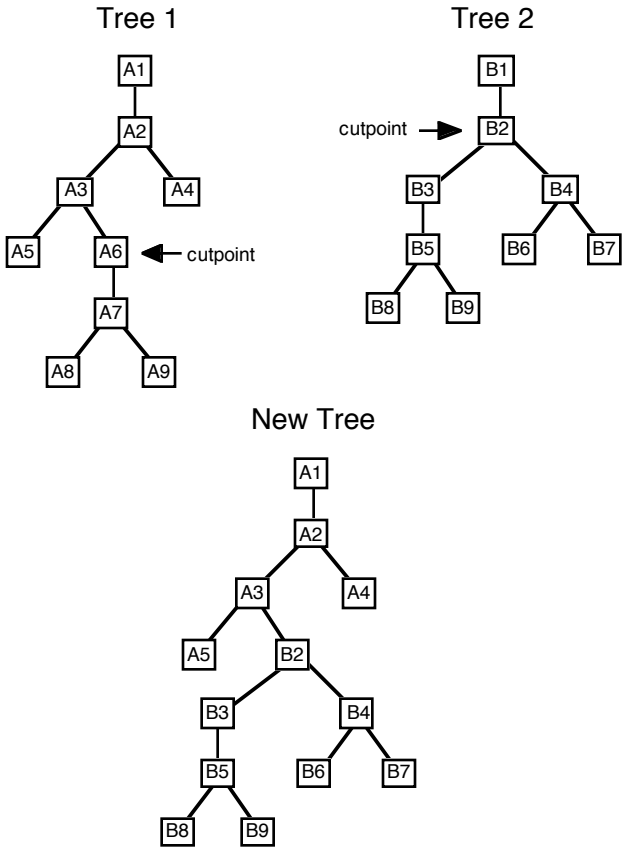

Figure 3. Illustration of the crossover operator.

The new tree, which was created by splicing together two trees in the current population, is then placed in the next generation.

Mutation Operator

The mutation operator removes part of an existing tree and replaces it with a randomly generated new part, using the same recursive function that created the first generation. A tree is selected from the current population and a cutpoint node is randomly selected among its nodes. That node and all beneath it are then removed. Subsequently, the recursive function is called at the cutpoint location to generate a new part of the tree. The modified tree is then placed in the next generation.

Solution Procedure

Figure 4 illustrates the basic solution procedure. The recursive function discussed previously creates the first generation. The fitness of each individual in the population is then determined followed by the initiation of a loop over generations. Within the generation loop, the next generation is created and the fitness of each individual within that generation is calculated. A test is then made to determine whether any individual meets a convergence criteria. If one is found, the loop is terminated and the calculation ended. If no individual is found, the calculation continues.

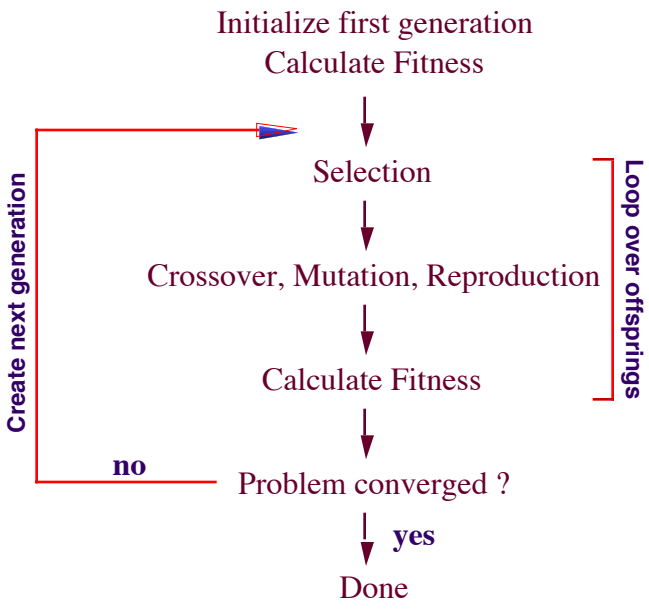

Figure 4. Solution procedure used by Paragon processors.

The calculations were done on the Intel massively parallel Paragon computer at Sandia National Laboratories. Running on the supercomputer required some slight modifications in the basic procedure to allow sharing of the "best" tree among processors. The number of processors allocated to a calculation was either 100 or 1800. A processor is identified by its number, whose range is 0 to the number of processors allocated minus 1. In our implementation, each processor had the same genetic program and ran independently of the other processors. At the end of each generation, each processor would determine the best tree in its population and send it to processor 0 where the globally best tree would be determined. Processor 0 would then broadcast the globally best tree to all processors. Accordingly, each processor would then decide if the globally best tree would be employed in creating the next generation, using an algorithm that depended on the generation number and its processor number. If a processor decided to use the globally best tree, it would do so only through mutation. That is, when mutation was selected, the processor would not use the selection operator to select an individual from the current population; rather, it would use the globally best tree.

The reason for not using the globally best tree all the time is to maintain diversity in the entire population. The convergence rate is proportional to a measure of the diversity. If the globally best tree was used by all processors at each generation, it would not be long before all of the trees would look much the same and the rate of improvement would be reduced.

Processor 0 performs two additional tasks. It writes the a binary representation of the globally best tree to disk for restart purposes, and it also writes an equivalent C source-code version that can be used in a simulator and in the actual robot.

Calculation Results

The calculation of the behavior was done on the Paragon in a series of four runs. Each of the first two runs used 100 processors and lasted 1 hour. Each of the last two runs used 1800 processors and lasted 6 hours. The globally best tree was then analyzed using a simulator that was similar to the one used in the genetic program to calculate fitness.

The final average root mean squared (RMS) distance of each robot to the target was found to be equal to about 2.2 for the 90 problems that were analyzed by the genetic program. The fitness for this tree was about 300 after the penalty term was subtracted. The tree had 484 nodes and was 24 levels deep. The distribution of node kinds is given in Figure 5, with the index numbers corresponding to those used in the table of functions and terminals presented previously. Note that the *subtract* node was used more often than others. We also found that the use of node kinds was related to information exchange

between the robots. Figure 6 illustrates a tree that was similar to the one generated in this study.

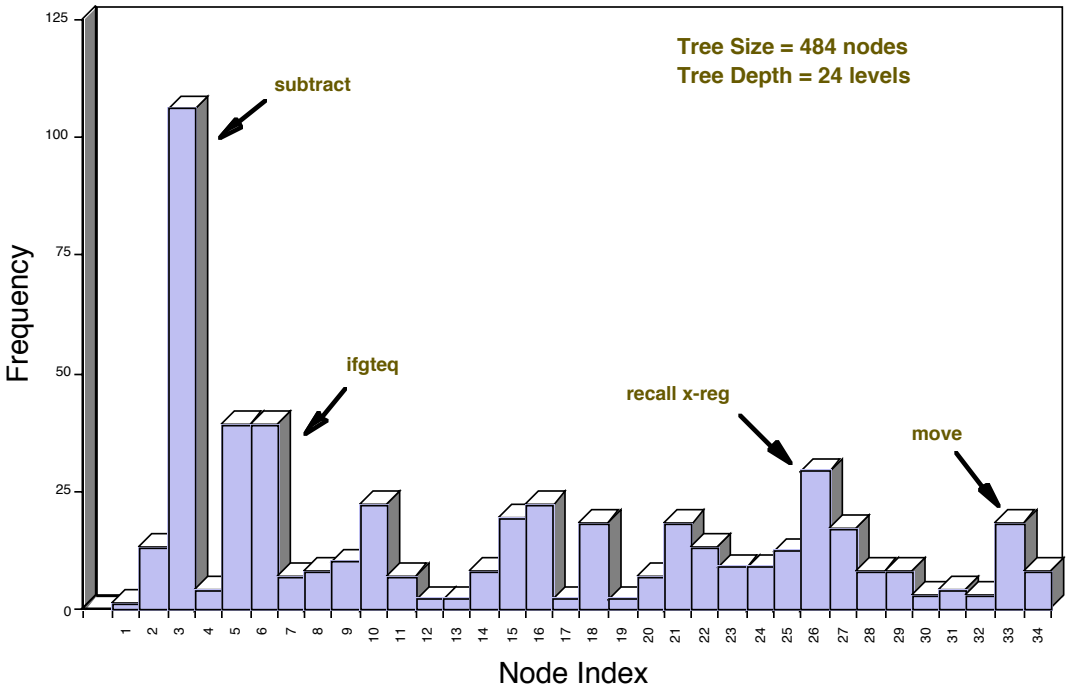

Figure 5. Distribution of the different kinds of nodes.

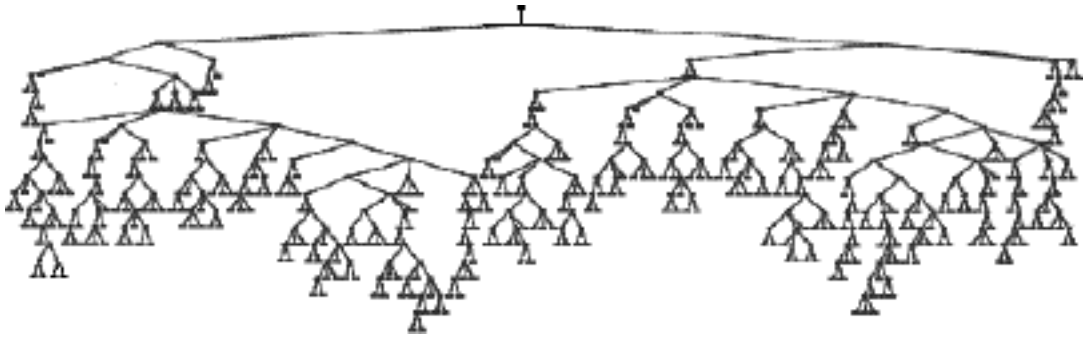

Figure 6. Illustration of a typical behavior tree found in this study.

We also observed that the final RMS distance to the target would abruptly decrease during the calculation. This phenomenon was attributed to the discovery of a new way to

find the target. We ran several additional calculations and observed the same phenomenon, as illustrated in Figure 7.

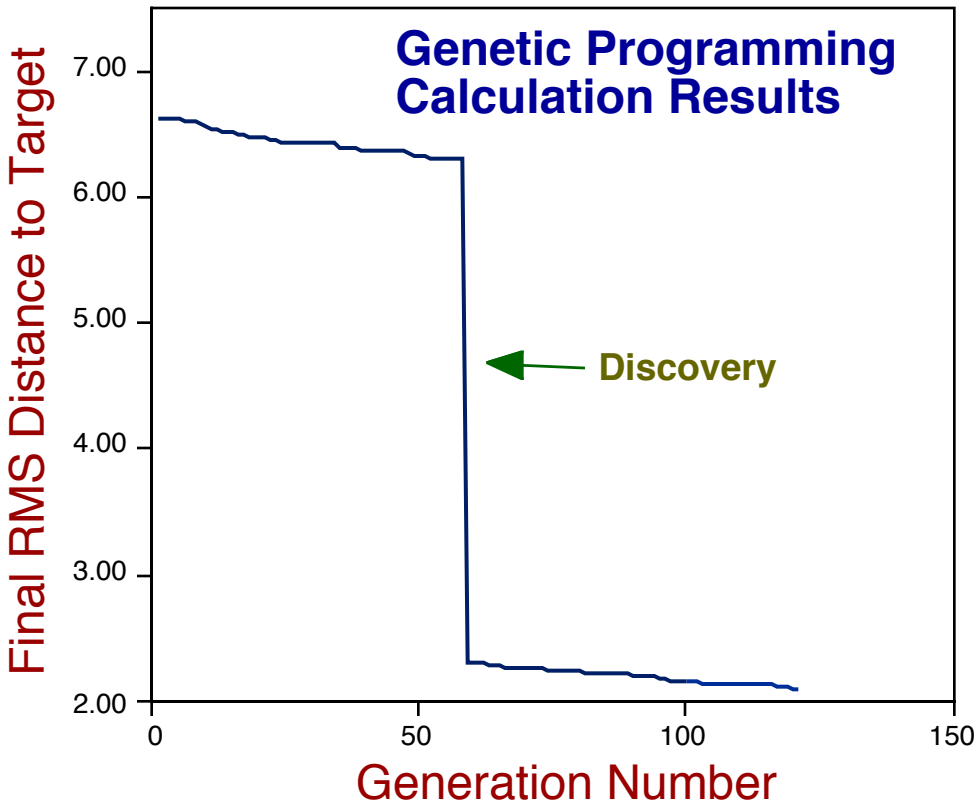

Figure 7. Final RMS distance by generation.

History of a Simulation

To highlight characteristics of the implementation, we have included a brief history of one simulation. In this simulation we used seven robots, one target, and two barriers. The initial positions were determined randomly. Figure 8 shows this history at 8 different steps during the transient. The squares in the figure are the robots, with the mark on each square denoting the direction that the robot is facing. The circle is the target, and the lines are barriers.

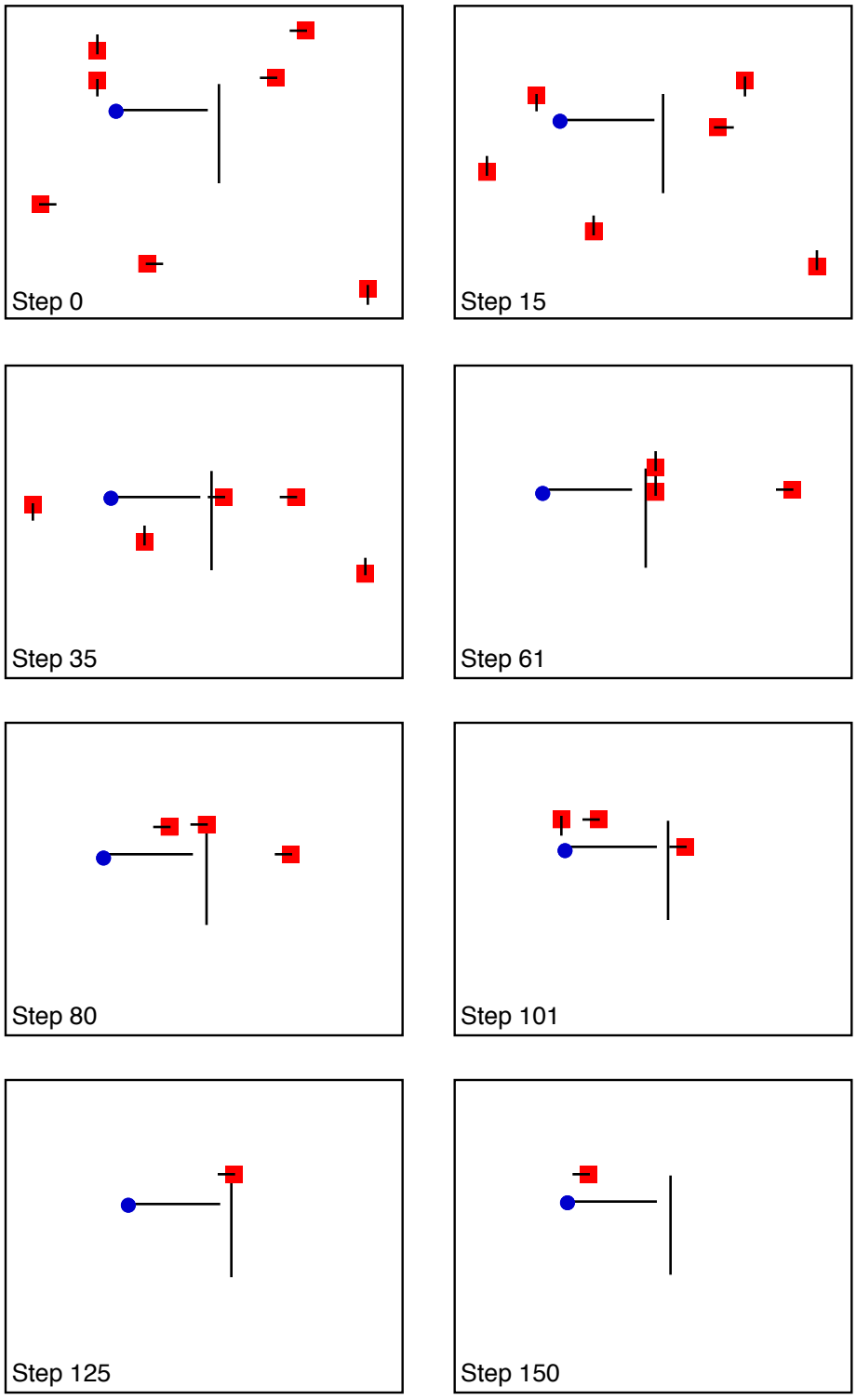

Figure 8. Results of a typical simulation calculation.

In the sample simulation, all robots found the target and none got trapped behind walls. We ran other cases where one or two robots did get trapped and therefore did not find the target. In some of those cases it appeared as though some of the robots helped the one that was trapped. We could not explain this cooperative behavior by studying the tree, which was too complex to analyze because of its size.

Table 2 lists the parameters used in the calculations of the sample simulation.

Table 2. Calculation Parameters

Parameter	Description	Value
FLDDIM1, FLDDIM2	Dimensions (width, length) of the field used in the simulations.	(100, 100)
MINTREESIZE	Minimum number of tree levels.	8
MAXTREESIZE	Maximum number of tree levels.	25
ALPHA	Penalty function constant, α.	1.0e-6
MAXSTEPS	Number of simulation steps performed per problem.	100
PROBSIZE	Number of problems run for fitness calculation.	90
POPSIZE	Number of trees on each processor of Paragon.	100
BUGSIZE	Number of robots used in simulation.	7
WALLSIZE	Number of walls used in simulation.	2
NTOUR	Number of individuals taking part in tournament selection.	4
PROBREP	Reproduction probability.	0.1
PROBCR	Crossover probability.	0.8
PROBMUT	Mutation probability.	0.1
NODEX	Number of Paragon processors - range (min, max).	(100, 1800)

Summary

The genetic programming model successfully produced a behavior for a collection of tracking robots. The computations were performed on Sandia's massively parallel Paragon computer at reasonable computing costs. Simulations confirmed that at least one robot located the target for a variety of starting conditions.

We have found that genetic programming models offer two advantages over more traditional methods for determining robotic behavior. The first, and perhaps most important, benefit is that new and sometimes novel solutions to problems are found—

ones that we might not have considered. This occurs because we do not constrain the solution. In genetic programming, we provide the tools (functions and terminals) and a goal to reach. The genetic program determines the best way to solve the problem. The second benefit is that the solutions appear to be more robust since many problem conditions and variations are investigated in the solution process.

Future Direction

The work on using genetic programming for robotic behavior is far from completion. More complex situations are likely to be encountered as robots enter actual field operations. Decisions beyond movement, such as concealment and operations in adverse conditions, must be considered.

We are also using this technology in war-gaming applications. Recently, we completed a simulation of two soldiers attacking a gun tower, where a genetic program was used to evolve successful strategies for the attacking soldiers. This work is continuing.

References

Holland, John H., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.

Goldberg, David E*., Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

Koza, John R., *Genetic Programming*, The MIT Press, 1992.